CSC 583 - Final Project - Carlos Ortiz

Data Collection, End-to-End Data Preparation & Vector Store Notebook, Custom Agentic

```
!pip install pandas numpy srt regex tqdm spacy rank-bm25 faiss-cpu chromadb openai python-dotenv kaggle pyarrow requests beautifulsoup4 lxml datasets
!python -m spacy download en_core_web_sm
!pip install langchain langchain-openai langchain-community sentence-transformers faiss-cpu rank-bm25
```

Show hidden output

COSINE of Thrones — End-to-End Data Preparation & Vector Store Notebook

This notebook builds the complete retrieval foundation for the **COSINE of Thrones** project.
It transforms raw Game of Thrones subtitles and character-lore datasets into a high-quality, augmented vector store used by the downstream RAG, evaluation, and agentic LangGraph-style modules.

The workflow performs four major phases:

1. Secure Environment Setup

Environment variables (OpenAI key, Kaggle credentials) are loaded using `python-dotenv`.
API keys are never hardcoded, ensuring the workflow is secure, portable, and version-controlled safely.

2. Data Acquisition (Kaggle + Hugging Face + Web Crawler)

- Downloads raw subtitle files (`season1.json` ... `season7.json`) from Kaggle.
- Parses and normalizes every subtitle line into a structured chronological DataFrame.
- Downloads the **Tuana/game-of-thrones** character-lore dataset from Hugging Face.
- Optionally crawls FunTrivia to collect human-written GOT QA for later evaluation.

3. Chunking & Dataset Augmentation

Subtitle lines are windowed into overlapping narrative chunks.
Lore entries are converted into parallel chunks with metadata (`chunk_kind = "character_lore"`).
The two sources are merged into **one unified dataset (`df_aug`)**.

4. Vector Store Construction (Embeddings + FAISS + BM25)

The notebook computes or loads:

- OpenAI text embeddings (`text-embedding-3-large`)
- A normalized FAISS index for dense semantic search
- A BM25 token-index for lexical matching
- A hybrid retrieval function that blends both signals

All artifacts are saved into `data/` so Phase 1-D can reload them instantly without recomputation.

Output of this notebook:
A complete, ready-to-query retrieval stack used by the COSINE of Thrones RAG pipeline, evaluation suite, and custom agentic graph.
This file is the foundation that powers all narrative-consistency experiments and downstream LangGraph-style workflows.

5. COSINE of Thrones — Custom Agentic RAG System

(Hand-Built **LangGraph-Style Pipeline)

In this notebook, we construct a **fully custom agentic RAG pipeline** inspired by LangGraph, but implemented entirely in plain Python — **no LangChain, no LangGraph, no framework dependencies**.

This system behaves like a miniature graph-orchestrated agent, with state flowing through a sequence of nodes that each perform a specific cognitive function. The design mirrors the way modern agentic LLM systems coordinate parsing, retrieval, reasoning, and synthesis, but is engineered from scratch to provide complete transparency and control.

What This Agentic Pipeline Does

The agent operates through four explicit "nodes", each a pure Python function:

1. **Query Parser**
   Uses spaCy to detect named entities and infer the question type (who/what/when/why).
   Annotates the shared state with structured metadata for downstream reasoning.
2. **Retriever**
   Invokes your upgraded `hybrid_search_aug()` function:
   a blend of FAISS vector search + BM25 lexical scoring + optional entity boosting
   over the merged subtitles + lore dataset (`df_aug`).
3. **Reranker (Optional)**
   If available, a cross-encoder (ms-marco-MiniLM) reorders retrieved chunks by semantic relevance.
   If not available, retrieval results flow through untouched.
4. **Synthesizer (LLM Answer Agent)**
   Formats evidence into a tightly controlled, citation-first prompt.
   Calls the generation model (e.g., `gpt-4o-mini`) and produces a grounded, one-sentence answer
   that must rely **only** on visible evidence.

The State Object (Your Own LangGraph Alternative)

A custom `RAGState` dataclass stores all intermediate values:

- the user question
- extracted entities
- question type
- retrieved chunks
- reranked chunks
- formatted evidence
- final answer
- per-node debug logs

This is analogous to LangGraph's "shared state passing" but implemented using clean Python dataclasses.

Manual Graph Runner

Your pipeline is executed via:

```
state = run_graph(question)
```

Environment Setup with python-dotenv and Kaggle Credentials

To keep API keys and dataset credentials secure, this notebook uses the python-dotenv package. The .env file stores sensitive values such as the OpenAI API key or Kaggle tokens so they never appear in source control.

The following snippet loads the .env file at runtime and injects the necessary environment variables into the notebook session. This approach keeps your workflow clean, portable, and compliant with best security practices.

```
# # Get Open AI key from .env
# import os
# os.environ["OPENAI_API_KEY"] = "your_openai_api_key_here"

import os
from dotenv import load_dotenv
load_dotenv()
os.environ["OPENAI_API_KEY"] = os.getenv("OPENAI_API_KEY")
```

Downloading the Game of Thrones Subtitle Dataset from Kaggle

This notebook pulls the Game of Thrones subtitle dataset directly from Kaggle.
To keep credentials secure, the workflow loads Kaggle variables from the `.env` file using `python-dotenv`.
The logic supports both a JSON-packed `KAGGLE_API_KEY` or the standard `KAGGLE_USERNAME` and `KAGGLE_KEY` format.

After loading the credentials, the Kaggle API authenticates and downloads the subtitle files into the `./data` directory.
These SRT files become the raw source material for all downstream preprocessing, cleaning, chunking, embedding, and RAG evaluation.

```
# Get from Kaggle - https://www.kaggle.com/datasets/gunnvant/game-of-thrones-srt
from dotenv import load_dotenv
import os, json
load_dotenv()  # call early

kaggle_api = os.getenv("KAGGLE_API_KEY")
if kaggle_api:
    try:
        creds = json.loads(kaggle_api)
        os.environ["KAGGLE_USERNAME"] = creds.get("username", os.getenv("KAGGLE_USERNAME"))
        os.environ["KAGGLE_KEY"] = creds.get("key", os.getenv("KAGGLE_KEY"))
    except Exception:
        # fall back to separate vars if JSON parse fails
        pass

# Ensure both vars exist
if not (os.getenv("KAGGLE_USERNAME") and os.getenv("KAGGLE_KEY")):
    raise EnvironmentError("Set KAGGLE_USERNAME and KAGGLE_KEY in .env")

import json
import pandas as pd
from kaggle.api.kaggle_api_extended import KaggleApi

# Initialize and authenticate the Kaggle API
api = KaggleApi()
try:
    api.authenticate()
except Exception as e:
    raise RuntimeError(f"Failed to authenticate Kaggle API: {e}")

# Define the Kaggle dataset path
dataset_path = "gunnvant/game-of-thrones-srt"  # Replace if needed

# Download and unzip the dataset to the ./data directory
os.makedirs("./data", exist_ok=True)
print(f"Downloading dataset '{dataset_path}' from Kaggle to ./data ...")
api.dataset_download_files(dataset_path, path="./data", unzip=True)
print("Dataset downloaded and unzipped successfully.")
```

```
Downloading dataset 'gunnvant/game-of-thrones-srt' from Kaggle to ./data ...
Dataset URL: https://www.kaggle.com/datasets/gunnvant/game-of-thrones-srt
Dataset downloaded and unzipped successfully.
```

```
# ==========================
# Phase 1 - Build Full Chunk Dataset (Subtitles + Lore)
# ==========================
import os, json, ast
from pathlib import Path
import pandas as pd
from tqdm import tqdm
from datasets import load_dataset

RAW_JSON_DIR = Path("data")  # directory containing season1.json ... season7.json

# ----------------------------------------------------------
# 1. Load subtitle JSON files into df_lines
# ----------------------------------------------------------

SE_RE = re.compile(r"[Ss](\d{1,2})[Ee](\d{1,2})")
```

```
def parse_episode_key(ep_key: str):
    """Extract S/E numbers from filenames like S01E04."""
    m = SE_RE.search(ep_key)
    season = int(m.group(1)) if m else None
    episode = int(m.group(2)) if m else None
    return season, episode

def load_kaggle_season_json(p: Path, seconds_per_line: float = 2.5) -> list[dict]:
    """Convert seasonN.json into line-level timestamped text rows."""
    with open(p, "r", encoding="utf-8") as f:
        data = json.load(f)  # {episode_filename: { "1": line1, "2": line2, ... }}

    rows = []
    for ep_key, lines_obj in data.items():
        season, episode = parse_episode_key(ep_key)

        # Sort numerically
        items = sorted(lines_obj.items(), key=lambda kv: int(kv[0]) if kv[0].isdigit() else kv[0])

        for idx, (_, text) in enumerate(items):
            if not text or not str(text).strip():
                continue

            t_start = idx * seconds_per_line
            t_end = t_start + seconds_per_line

            rows.append({
                "season": season,
                "episode": episode,
                "t_start": float(t_start),
                "t_end": float(t_end),
                "text": str(text).strip()
            })

    return rows

# Ingest all season JSON files
all_lines = []
for p in sorted(RAW_JSON_DIR.glob("season*.json")):
    print(f"📥 Loading {p.name} ...")
    all_lines.extend(load_kaggle_season_json(p))

df_lines = pd.DataFrame(all_lines).dropna(subset=["text"])
df_lines = df_lines.sort_values(["season","episode","t_start"]).reset_index(drop=True)
print(f"🎬 Loaded {len(df_lines)} subtitle lines")


# -----------------------------------------------------------
# 2. Build subtitle chunks from df_lines
# -----------------------------------------------------------

def build_subtitle_chunks(df_lines, window=5, stride=3):
    rows = []
    for (season, episode), group in df_lines.groupby(["season", "episode"]):
        texts  = group["text"].tolist()
        starts = group["t_start"].tolist()
        ends   = group["t_end"].tolist()

        for i in range(0, len(texts), stride):
            chunk_text = " ".join(texts[i:i+window]).strip()
            if not chunk_text:
                continue

            chunk_start = starts[i]
            chunk_end   = ends[min(i + window - 1, len(ends) - 1)]

            rows.append({
                "season": season,
                "episode": episode,
                "t_start": float(chunk_start),
                "t_end": float(chunk_end),
                "text": chunk_text,
                "chunk_kind": "subtitle"
            })

    df_chunks = pd.DataFrame(rows).dropna(subset=["text"])
    print(f"🧱 Built {len(df_chunks)} subtitle chunks")
    return df_chunks

df_chunks = build_subtitle_chunks(df_lines)


# ------------------------------------------------------------
# 3. Load Hugging Face character lore dataset
# ------------------------------------------------------------

print("\n📥 Loading Tuana/game-of-thrones (character lore)...")
ds = load_dataset("Tuana/game-of-thrones", split="train")
df_lore = ds.to_pandas()

def extract_name(meta):
    if isinstance(meta, str):
        try:
            meta = ast.literal_eval(meta)
        except:
            return None
    if isinstance(meta, dict):
        return meta.get("name", None)
    return None

df_lore["meta_name"] = df_lore["meta"].apply(extract_name)
df_lore = df_lore.rename(columns={"content": "text"})
df_lore = df_lore.dropna(subset=["text"]).reset_index(drop=True)

df_lore_chunks = pd.DataFrame({
    "season": None,
    "episode": None,
    "t_start": 0.0,
    "t_end": 0.0,
    "text": df_lore["text"].astype(str).str.strip(),
    "chunk_kind": "character_lore"
})

print(f"📚 Built {len(df_lore_chunks)} lore chunks")


# ------------------------------------------------------------
# 4. Merge subtitle + lore chunks
# ------------------------------------------------------------

df_aug = pd.concat([df_chunks, df_lore_chunks], ignore_index=True)
df_aug = df_aug.dropna(subset=["text"]).reset_index(drop=True)

print(f"\n🔗 Merged dataset now has {len(df_aug)} total chunks")


# ------------------------------------------------------------
# 5. Save snapshot for Phase 1-C (embeddings + FAISS)
# ------------------------------------------------------------

OUT = Path("data/got_augmented_lore.csv")
OUT.parent.mkdir(parents=True, exist_ok=True)
df_aug.to_csv(OUT, index=False)
print(f"💾 Saved enriched + chunked dataset → {OUT}")
```

```
📥 Loading season1.json ...
📥 Loading season2.json ...
📥 Loading season3.json ...
📥 Loading season4.json ...
📥 Loading season5.json ...
📥 Loading season6.json ...
📥 Loading season7.json ...
```

```
🎬 Loaded 44890 subtitle lines
🧱 Built 14990 subtitle chunks

📥 Loading Tuana/game-of-thrones (character lore)...
📚 Built 2357 lore chunks

🔗 Merged dataset now has 17347 total chunks
💾 Saved enriched + chunked dataset → data/got_augmented_lore.csv
```

Parsing and Normalizing Kaggle Subtitle JSON Files

This notebook uses a helper script (`ingest_kaggle_json.py`) to convert the raw Kaggle subtitle JSON files into a clean, structured DataFrame. The raw dataset stores each episode as a large dictionary of numbered lines, so this step extracts season and episode identifiers, orders the lines correctly, and synthesizes simple timestamp fields to support downstream chunking and embedding steps.

Each `season*.json` file is parsed into rows containing the season, episode, start time, end time, and the cleaned subtitle text. All seasons are merged into a single DataFrame, sorted chronologically, and filtered to keep Seasons 1 through 7. The resulting dataset becomes the standardized foundation for all further preprocessing, retrieval indexing, and narrative evaluation.

```python
# ingest_kaggle_json.py
from pathlib import Path
import json, regex as re
import pandas as pd

RAW_JSON_DIR = Path("data")  # put season1.json ... season7.json here

# Extract S/E from "Game Of Thrones S01E01 Winter Is Coming.srt"
SE_RE = re.compile(r"[Ss](\d{1,2})[Ee](\d{1,2})")

def parse_episode_key(ep_key: str):
    m = SE_RE.search(ep_key)
    season = int(m.group(1)) if m else None
    episode = int(m.group(2)) if m else None
    return season, episode

def load_kaggle_season_json(p: Path, seconds_per_line: float = 2.5) -> list[dict]:
    with open(p, "r", encoding="utf-8") as f:
        data = json.load(f)  # dict: episode_filename -> { "1": "line", "2": "line", ... }

    rows = []
    for ep_key, lines_obj in data.items():
        season, episode = parse_episode_key(ep_key)
        # lines_obj keys are strings of integers; sort numerically
        line_items = sorted(lines_obj.items(), key=lambda kv: int(kv[0]) if kv[0].isdigit() else kv[0])
        for idx, (_, text) in enumerate(line_items):
            if not text or not str(text).strip():
                continue
            # synthesize simple timestamps so downstream chunkers work
            t_start = idx * seconds_per_line
            t_end   = t_start + seconds_per_line
            rows.append({
                "season": season,
                "episode": episode,
                "t_start": float(t_start),
                "t_end": float(t_end),
                "text": str(text).strip()
            })
    return rows

# ---- Ingest all season*.json files ----
all_rows = []
for p in sorted(RAW_JSON_DIR.glob("season*.json")):
    all_rows.extend(load_kaggle_season_json(p, seconds_per_line=2.5))

df_lines = pd.DataFrame(all_rows).dropna(subset=["text"])
# keep S1–S7 if desired
df_lines = df_lines[(df_lines["season"] >= 1) & (df_lines["season"] <= 7)]
df_lines = df_lines.sort_values(["season", "episode", "t_start"]).reset_index(drop=True)
```

```
print("Loaded lines:", len(df_lines))
print(df_lines.head())
```

```
Loaded lines: 44890
   season  episode  t_start  t_end  \
0       1        1      0.0    2.5
1       1        1      2.5    5.0
2       1        1      5.0    7.5
3       1        1      7.5   10.0
4       1        1     10.0   12.5

                                              text
0                                       Easy, boy.
1               What do you expect? They're savages.
2            One lot steals a goat from another lot,
3  before you know it they're ripping each other ...
4    I've never seen wildlings do a thing like this.
```

FunTrivia Web Crawler: Collecting External Game of Thrones QA Pairs

This section uses a robust web crawler to gather high-quality Game of Thrones trivia questions from FunTrivia.com.
It extracts structured fields such as the question number, question text, short answer, explanation, and the source URL.

The crawler performs the following steps:

1. **HTML Parsing of Question Blocks**
   Each page is scanned for structured `schema.org/Question` entries. The parser extracts the question text, the accepted answer, and any explanatory text, handling many variations in page formatting.
2. **Iterative Pagination Across Trivia Pages**
   The crawler automatically identifies “next page” links using multiple fallback strategies. This ensures reliable traversal across FunTrivia’s multi-page trivia sets, even when pagination markup varies.
3. **Duplicate Prevention and Safety Guards**
   Each (qnum, question) pair is tracked to avoid duplicates. The crawler enforces limits on maximum questions, maximum pages, and includes polite delays between requests.
4. **Dataset Assembly and Export**
   All extracted records are compiled into a DataFrame and saved as
   `data/funtrivia_questions_all_200.csv`.
   These external QA pairs are later used for narrative evaluation, cross-checking retrieval, or augmenting the golden dataset.

This crawler enables the notebook to incorporate additional grounded, human-written GOT questions that serve as a valuable benchmark for retrieval and answer-generation quality.

```
# Jupyter cell: robust FunTrivia GOT crawler — extracts qnum, question, short answer, explanation, source_url

import requests, time, urllib.parse
from bs4 import BeautifulSoup
from pathlib import Path
import pandas as pd
from tqdm import tqdm

HEADERS = {
    "User-Agent": "Mozilla/5.0 (compatible; funtrivia-scraper/1.0; +https://github.com/you)",
    "Accept-Language": "en-US,en;q=0.9",
}
BASE = "https://www.funtrivia.com"
START = "https://www.funtrivia.com/en/Television/Game-of-Thrones-20275.html"

def parse_questions_from_soup(soup, source_url):
    out = []
    for q_div in soup.find_all(attrs={"itemtype":"http://schema.org/Question"}):
        # number
        step = q_div.find(class_="step")
        qnum = step.get_text(strip=True) if step else None

        # question text (itemprop=name)
        qname = q_div.find(attrs={"itemprop":"name"})
        question = qname.get_text(" ", strip=True) if qname else None

        # accepted answer block
        ans_bq = q_div.find("blockquote", class_="answer")
        short_ans = ""
        explanation = ""
        if ans_bq:
            # find all itemprop="text" elements inside blockquote (first often contains short answer, second explanation)
            texts = ans_bq.find_all(attrs={"itemprop":"text"})
            if texts:
                # Prefer bold inside first text for concise answer
                first = texts[0]
                b = first.find("b")
                if b and b.get_text(strip=True):
                    short_ans = b.get_text(strip=True)
                else:
                    # fallback: strip "Answer:" prefix if present
                    t0 = first.get_text(" ", strip=True)
                    short_ans = t0.replace("Answer:", "").strip()
                if len(texts) > 1:
                    explanation = texts[1].get_text(" ", strip=True)
                else:
                    # sometimes explanation is directly after the bold inside same node
                    # try to remove the bold text from first node and use remaining as explanation
                    if b:
                        # remove bold tag content to get any trailing explanation text
                        for tag in first.find_all("b"):
                            tag.decompose()
                        rem = first.get_text(" ", strip=True)
                        if rem:
                            explanation = rem
            else:
                # last-resort: take blockquote text and try to split first sentence as short answer
                full = ans_bq.get_text(" ", strip=True)
                if ":" in full:
                    # common "Answer: X Explanation..."
                    parts = full.split(":", 1)
                    short_ans = parts[1].split()[0]
                    explanation = parts[1].strip()
                else:
                    short_ans = full
        out.append({
            "qnum": qnum,
            "question": question,
            "answer_short": short_ans,
            "explanation": explanation,
            "source_url": source_url
        })
    return out

def crawl_funtrivia(start_url=START, max_q=200, max_pages=50, pause=0.9):
    seen_q = set()
    results = []
    url = start_url
    pages = 0
    while url and pages < max_pages and len(results) < max_q:
        pages += 1
        try:
            resp = requests.get(url, headers=HEADERS, timeout=20)
            resp.raise_for_status()
        except Exception as e:
            print(f"failed {url}: {e}")
            break
        soup = BeautifulSoup(resp.text, "lxml")
        parsed = parse_questions_from_soup(soup, url)
        for item in parsed:
            key = (item["qnum"], item["question"])
            if key in seen_q:
                continue
            seen_q.add(key)
            results.append(item)
            if len(results) >= max_q:
                break

        # find next page (link rel="next" or pager anchors)
        next_link = None
        link_tag = soup.find("link", rel="next")
        if link_tag and link_tag.get("href"):
            next_link = urllib.parse.urljoin(BASE, link_tag["href"])
        else:
            # fallback: look for anchor with pattern _2.html etc.
            pager = soup.select_one(".pagelist, .pagelinks, .pages")
            if pager:
                a_next = pager.find("a", string=lambda s: s and s.strip().isdigit() and int(s.strip()) == pages+1)
                if a_next and a_next.get("href"):
                    next_link = urllib.parse.urljoin(BASE, a_next["href"])
            # last fallback: look for any "Next" text
            if not next_link:
                a_next2 = soup.find("a", string=lambda s: s and "next" in s.lower())
                if a_next2 and a_next2.get("href"):
                    next_link = urllib.parse.urljoin(BASE, a_next2["href"])

        url = next_link
        time.sleep(pause)

    df = pd.DataFrame(results)
    return df

# run crawler (adjust max_q as needed)
df_all = crawl_funtrivia(max_q=200, max_pages=40, pause=0.8)

outdir = Path("data")
outdir.mkdir(parents=True, exist_ok=True)
outpath = outdir / "funtrivia_questions_all_200.csv"
df_all.to_csv(outpath, index=False)
print(f"Scraped {len(df_all)} QA pairs -> {outpath}")
df_all.head()
#```# filepath: /Users/carlosrortiz/Documents/csc5830-ThroneRag/got_eda.ipynb
# Jupyter cell: robust FunTrivia GOT crawler — extracts qnum, question, short answer, explanation, source_url
!pip install -q requests beautifulsoup4 lxml tqdm

import requests, time, urllib.parse
from bs4 import BeautifulSoup
from pathlib import Path
import pandas as pd
from tqdm import tqdm

HEADERS = {
    "User-Agent": "Mozilla/5.0 (compatible; funtrivia-scraper/1.0; +https://github.com/you)",
    "Accept-Language": "en-US,en;q=0.9",
}
BASE = "https://www.funtrivia.com"
START = "https://www.funtrivia.com/en/Television/Game-of-Thrones-20275.html"

def parse_questions_from_soup(soup, source_url):
    out = []
    for q_div in soup.find_all(attrs={"itemtype":"http://schema.org/Question"}):
        # number
        step = q_div.find(class_="step")
        qnum = step.get_text(strip=True) if step else None

        # question text (itemprop=name)
        qname = q_div.find(attrs={"itemprop":"name"})
        question = qname.get_text(" ", strip=True) if qname else None

        # accepted answer block
        ans_bq = q_div.find("blockquote", class_="answer")
        short_ans = ""
        explanation = ""
        if ans_bq:
            # find all itemprop="text" elements inside blockquote (first often contains short answer, second explanation)
```

```
            texts = ans_bq.find_all(attrs={"itemprop":"text"})
            if texts:
                # Prefer bold inside first text for concise answer
                first = texts[0]
                b = first.find("b")
                if b and b.get_text(strip=True):
                    short_ans = b.get_text(strip=True)
                else:
                    # fallback: strip "Answer:" prefix if present
                    t0 = first.get_text(" ", strip=True)
                    short_ans = t0.replace("Answer:", "").strip()
                if len(texts) > 1:
                    explanation = texts[1].get_text(" ", strip=True)
                else:
                    # sometimes explanation is directly after the bold inside same node
                    # try to remove the bold text from first node and use remaining as explanation
                    if b:
                        # remove bold tag content to get any trailing explanation text
                        for tag in first.find_all("b"):
                            tag.decompose()
                        rem = first.get_text(" ", strip=True)
                        if rem:
                            explanation = rem
            else:
                # last-resort: take blockquote text and try to split first sentence as short answer
                full = ans_bq.get_text(" ", strip=True)
                if ":" in full:
                    # common "Answer: X Explanation..."
                    parts = full.split(":", 1)
                    short_ans = parts[1].split()[0]
                    explanation = parts[1].strip()
                else:
                    short_ans = full
        out.append({
            "qnum": qnum,
            "question": question,
            "answer_short": short_ans,
            "explanation": explanation,
            "source_url": source_url
        })
    return out

def crawl_funtrivia(start_url=START, max_q=200, max_pages=50, pause=0.9):
    seen_q = set()
    results = []
    url = start_url
    pages = 0
    while url and pages < max_pages and len(results) < max_q:
        pages += 1
        try:
            resp = requests.get(url, headers=HEADERS, timeout=20)
            resp.raise_for_status()
        except Exception as e:
            print(f"failed {url}: {e}")
            break
        soup = BeautifulSoup(resp.text, "lxml")
        parsed = parse_questions_from_soup(soup, url)
        for item in parsed:
            key = (item["qnum"], item["question"])
            if key in seen_q:
                continue
            seen_q.add(key)
            results.append(item)
            if len(results) >= max_q:
                break

        # find next page (link rel="next" or pager anchors)
        next_link = None
        link_tag = soup.find("link", rel="next")
        if link_tag and link_tag.get("href"):
            next_link = urllib.parse.urljoin(BASE, link_tag["href"])
        else:
            # fallback: look for anchor with pattern _2.html etc.
            pager = soup.select_one(".pagelist, .pagelinks, .pages")
            if pager:
                a_next = pager.find("a", string=lambda s: s and s.strip().isdigit() and int(s.strip()) == pages+1)
                if a_next and a_next.get("href"):
                    next_link = urllib.parse.urljoin(BASE, a_next["href"])
            # last fallback: look for any "Next" text
            if not next_link:
                a_next2 = soup.find("a", string=lambda s: s and "next" in s.lower())
                if a_next2 and a_next2.get("href"):
                    next_link = urllib.parse.urljoin(BASE, a_next2["href"])

        url = next_link
        time.sleep(pause)

    df = pd.DataFrame(results)
    return df

# run crawler (adjust max_q as needed)
df_all = crawl_funtrivia(max_q=200, max_pages=40, pause=0.8)

outdir = Path("data")
outdir.mkdir(parents=True, exist_ok=True)
outpath = outdir / "funtrivia_questions_all_200.csv"
df_all.to_csv(outpath, index=False)
print(f"Scraped {len(df_all)} QA pairs -> {outpath}")
df_all.head()
```

```
Scraped 165 QA pairs -> data/funtrivia_questions_all_200.csv
Scraped 165 QA pairs -> data/funtrivia_questions_all_200.csv
```

	qnum	question	answer_short	explanation	source_url
0	1	Who are the two brothers of the Night's Watch ...	Brant and Derek	Sam is beaten half to death trying to defend G...	https://www.funtrivia.com/en/Television/Game-o...
1	2	Daenerys is traded away as a bride to a "savag...	Viserys	Prince Viserys (Harry Lloyd) is the older and ...	https://www.funtrivia.com/en/Television/Game-o...
2	3	The Night King has so far been the main villai...	Viserion	The Night King kills Viserion with a fantastic...	https://www.funtrivia.com/en/Television/Game-o...
3	4	At the conclusion of season 7, Cersei Lanniste...	The Golden Company	The Golden Company is an an elite group of sel...	https://www.funtrivia.com/en/Television/Game-o...
4	5	What is the name of Jon's direwolf?	Ghost	Ghost, unlike the other direwolves found by th...	https://www.funtrivia.com/en/Television/Game-o...

Next steps: Generate code with `df_all` New interactive sheet

One-Time Retrieval Setup: Embeddings, FAISS Indexing, BM25, and Hybrid Search

This section initializes the full retrieval backend used throughout the notebook.
It runs only once per session, or reruns automatically if the underlying chunk dataframe changes.

The setup performs four major tasks:

1. **Embedding Generation (or Loading Artifacts)**
   The system uses a single embedding model (`text-embedding-3-large`) and generates vector embeddings for all text chunks.
   If previously saved vector files or FAISS indexes are found in the `data/` directory, these are loaded instead of recomputed.
2. **FAISS Index Construction**
   The vectors are normalized and stored in a FAISS inner-product index for efficient semantic search.
   If an index already exists, it is loaded and validated. If loading fails, the index is rebuilt automatically.
3. **BM25 Token-Based Retrieval**
   Each chunk is tokenized and indexed using BM25 (RankBM25). This provides a complementary sparse retrieval pathway that helps catch lexical matches missed by embeddings.
4. **Hybrid Search Initialization**
   A single `hybrid_search()` function combines FAISS similarity scores and BM25 scores to produce a ranked list of candidate chunks.
   It includes safety checks for index length mismatches, truncated vectors, missing embeddings, and out-of-range FAISS hits.

Together, these components form the notebook's unified retrieval layer, enabling fast, stable, and repeatable semantic + lexical search over the Game of Thrones subtitle corpus.

```
# ==========================
# Phase 1–C — Embeddings, FAISS, BM25
# ==========================
import numpy as np
import pandas as pd
import pickle
from pathlib import Path
from tqdm import tqdm

# Prerequisites:
# Requires df_aug from Phase 1–B
if "df_aug" not in globals():
    raise RuntimeError("df_aug not found. Run Phase 1–B before Phase 1–C.")

BASE = Path("data")
BASE.mkdir(exist_ok=True)

# Artifact paths
VEC_PATH   = BASE / "got_aug_vecs.npy"
INDEX_PATH = BASE / "got_aug_faiss.bin"
TOK_PATH   = BASE / "got_aug_corpus_tokens.pkl"

# ------------------------------------------
# OpenAI setup
# ------------------------------------------
from openai import OpenAI
import os

OPENAI_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_KEY:
    raise RuntimeError("OPENAI_API_KEY missing. Load .env or set manually.")

client = OpenAI(api_key=OPENAI_KEY)
EMBED_MODEL = "text-embedding-3-large"

# ------------------------------------------
# FAISS
# ------------------------------------------
import faiss

# ------------------------------------------
# spaCy for tokenization (BM25)
# ------------------------------------------
import spacy
try:
    nlp = spacy.load("en_core_web_sm")
except:
    raise RuntimeError("spaCy model missing. Run: python -m spacy download en_core_web_sm")

# ------------------------------------------
# BM25
# ------------------------------------------
from rank_bm25 import BM25Okapi

# ------------------------------------------
# Helper embedder (safe batching)
# ------------------------------------------
def embed_texts(texts, batch_size=16, max_chars=8000):
    vecs = []
    for i in tqdm(range(0, len(texts), batch_size), desc="Embedding"):
        batch = texts[i:i+batch_size]
        batch_trunc = [
            t if len(t) <= max_chars else t[:max_chars] + " [TRUNCATED]"
            for t in batch
        ]
        try:
            resp = client.embeddings.create(model=EMBED_MODEL, input=batch_trunc)
            vecs.extend([e.embedding for e in resp.data])
        except Exception as e:
            print(f"Batch {i} failed: {e}. Retrying per item...")
            for t in batch_trunc:
                for attempt in range(3):
                    try:
                        r = client.embeddings.create(model=EMBED_MODEL, input=[t])
                        vecs.extend([e.embedding for e in r.data])
                        break
                    except Exception:
                        time.sleep(1 + attempt)
                else:
                    raise RuntimeError("Failed embedding despite retries.")
    return np.asarray(vecs, dtype="float32")


# ------------------------------------------
# Step 1 — Build/load embedding vectors
# ------------------------------------------
if VEC_PATH.exists():
    print(f"📥 Loading vectors: {VEC_PATH}")
    vecs = np.load(VEC_PATH)
else:
    print("🔮 Computing embeddings for df_aug ...")
    texts = df_aug["text"].fillna("").tolist()
    vecs = embed_texts(texts)
    np.save(VEC_PATH, vecs)
    print(f"💾 Saved embeddings → {VEC_PATH}")

# ------------------------------------------
# Step 2 — Build/load FAISS index
# ------------------------------------------
if INDEX_PATH.exists():
    print(f"📥 Loading FAISS index: {INDEX_PATH}")
    try:
        index = faiss.read_index(str(INDEX_PATH))
    except Exception as e:
        print(f"⚠️ Failed to load FAISS index ({e}). Rebuilding...")
        faiss.normalize_L2(vecs)
        index = faiss.IndexFlatIP(vecs.shape[1])
        index.add(vecs)
        faiss.write_index(index, str(INDEX_PATH))
else:
    print("⚙️ Building FAISS index...")
    faiss.normalize_L2(vecs)
    index = faiss.IndexFlatIP(vecs.shape[1])
    index.add(vecs)
    faiss.write_index(index, str(INDEX_PATH))
    print(f"💾 Saved FAISS index → {INDEX_PATH}")

print(f"FAISS ready → {index.ntotal} vectors")

# ------------------------------------------
# Step 3 — Build/load BM25 token cache
# ------------------------------------------
if TOK_PATH.exists():
    print(f"📥 Loading BM25 tokens: {TOK_PATH}")
    with open(TOK_PATH, "rb") as f:
        corpus_tokens = pickle.load(f)
else:
    print("🧠 Tokenizing for BM25 ...")
    corpus_tokens = [
        [tok.text.lower() for tok in nlp(txt)]
        for txt in tqdm(df_aug["text"].fillna(""), desc="BM25 Tokenizing")
    ]
    with open(TOK_PATH, "wb") as f:
        pickle.dump(corpus_tokens, f)
    print(f"💾 Saved BM25 tokens → {TOK_PATH}")

bm25 = BM25Okapi(corpus_tokens)
print("BM25 ready.")
```

```
# -----------------------------------------
# Final outputs of Phase 1-C
# -----------------------------------------
print("\n🎉 Phase 1-C complete.")
print("Artifacts available:")
print(" - df_aug")
print(f" - {VEC_PATH.name}")
print(f" - {INDEX_PATH.name}")
print(f" - {TOK_PATH.name}")
print("\nYou may now run Phase 1-D (fast loader) or your LangGraph nodes.")
```

```
📥 Loading vectors: data/got_aug_vecs.npy
📥 Loading FAISS index: data/got_aug_faiss.bin
FAISS ready → 17347 vectors
📥 Loading BM25 tokens: data/got_aug_corpus_tokens.pkl
BM25 ready.

🎉 Phase 1-C complete.
Artifacts available:
 - df_aug
 - got_aug_vecs.npy
 - got_aug_faiss.bin
 - got_aug_corpus_tokens.pkl

You may now run Phase 1-D (fast loader) or your LangGraph nodes.
```

Phase 1-D — Fast Loader (Skip Embedding, Skip Tokenization, Skip FAISS Builds)

This cell is a **quick-start environment loader**.
It loads all the *precomputed* artifacts produced earlier in Phase 1-C so you can immediately start running the RAG pipeline, LangGraph-style agent, evaluations, or UI tests **without spending time recomputing embeddings or building FAISS/BM25**.

You only need to run Phase 1-C **once** (or whenever the dataset changes).
After that, this loader cell acts as your **instant startup**.

What This Cell Does

1. Loads All Retrieval Artifacts from Disk

The cell loads (whichever version exists):

- `df_aug` (combined subtitles + lore chunks)
- `got_aug_vecs.npy` (embeddings)
- `got_aug_faiss.bin` (FAISS index)
- `got_aug_corpus_tokens.pkl` (BM25 tokenized corpus)

It chooses the best available file using path fallbacks, so you never have to manually change paths.

---

2. Reconstructs All Retrieval Objects in Memory

Once artifacts are loaded from disk, it reconstructs:

- The **FAISS index**
- The **BM25Okapi** scorer
- The **hybrid_search_aug** retrieval function
- The OpenAI client (`client`)
- The spaCy NLP pipeline (`nlp`)

This produces a full, ready-to-use retrieval stack **exactly like the state produced by running Phase 1-C**.

---

3. Guards Against Missing Artifacts

If any critical file is missing (DF, vectors, FAISS, BM25), the loader throws **clear diagnostic errors** so you know which Phase 1-C step must be rerun.

This avoids silent failures and prevents mismatched vector length issues.

---

4. Re-registers Required Global Variables

The notebook cells that follow expect the following to exist:

- `df_aug`
- `index`
- `bm25`
- `client`
- `nlp`
- `EMBED_MODEL`

This loader explicitly sets all these globals so the rest of the notebook works without modification.

---

5. Optional Cross-Encoder Reranker

If available, it loads the sentence-transformers cross-encoder:

```
# language: python
# Quick "load everything" helper to skip embedding / tokenizing steps.
# Run this cell before the nodes cell.

import os, pickle
from pathlib import Path
import numpy as np
import pandas as pd

# ...existing code...
def hybrid_search_aug(query: str, topk: int = 10, alpha: float = 0.35, cand_mult: int = 20):
    import numpy as np, pandas as pd
    # require loader cell to have set: df_aug, index, bm25, client, nlp, EMBED_MODEL
    if "df_aug" not in globals():
        raise NameError("df_aug not found. Run the loader cell that loads precomputed artifacts first.")
    q_emb = client.embeddings.create(model=EMBED_MODEL, input=[query]).data[0].embedding
    qv = np.asarray(q_emb, dtype="float32")[None, :]
    faiss.normalize_L2(qv)
    D, I = index.search(qv, topk * cand_mult)
    vec_scores, vec_idx = D[0].tolist(), I[0].tolist()

    q_tokens = [t.text.lower() for t in nlp(query)]
    bm_scores = bm25.get_scores(q_tokens)
    bm_top = np.argsort(bm_scores)[::-1][:topk * cand_mult]

    max_valid = len(df_aug)
    valid_vec_pairs = [
        (int(idx), float(score))
        for idx, score in zip(vec_idx, vec_scores)
        if isinstance(idx, (int, np.integer)) and 0 <= int(idx) < max_valid
    ]
    v_map = {i: s for i, s in valid_vec_pairs}
    vec_idx_valid = [i for i, _ in valid_vec_pairs]
    bm_top_valid = [int(i) for i in bm_top if 0 <= int(i) < max_valid]

    cand = list(set(vec_idx_valid) | set(bm_top_valid))
    if not cand:
        return pd.DataFrame([])

    bm_max = max(bm_scores) if len(bm_scores) else 1.0
    scored = []
    for i in cand:
        v = float(v_map.get(int(i), 0.0))
        b = float(bm_scores[int(i)]) / (bm_max + 1e-6)
        scored.append((int(i), alpha * v + (1 - alpha) * b))
    scored.sort(key=lambda x: x[1], reverse=True)

    rows = []
```

```python
    for i, sc in scored[:topk]:
        r = df_aug.iloc[int(i)].to_dict()
        r["score"] = float(sc)
        rows.append(r)
    return pd.DataFrame(rows)
# ...existing code...

# OpenAI client (same API used in notebook)
try:
    from openai import OpenAI
except Exception:
    raise RuntimeError("openai package or OpenAI import not available. Install 'openai' and retry.")
OPENAI_KEY = os.getenv("OPENAI_API_KEY")
if not OPENAI_KEY:
    raise RuntimeError("OPENAI_API_KEY not set in environment. Set it or run dotenv load step.")
client = OpenAI(api_key=OPENAI_KEY)

# spaCy
try:
    import spacy
    nlp = spacy.load("en_core_web_sm")
except Exception as e:
    raise RuntimeError(f"spaCy model not ready: {e}. Run `python -m spacy download en_core_web_sm` if needed.")

# BM25 and FAISS
try:
    from rank_bm25 import BM25Okapi
except Exception:
    raise RuntimeError("rank_bm25 not installed. pip install rank-bm25")

try:
    import faiss
except Exception:
    raise RuntimeError("faiss-cpu not installed. pip install faiss-cpu")

# Candidate artifact paths (prefer augmented artifacts)
BASE = Path("data")
DF_AUG_PATHS = [BASE / "got_aug_chunks.csv", BASE / "got_augmented_lore.csv", BASE / "got_aug_chunks.csv"]
DF_CHUNK_PATHS = [BASE / "got_chunks.csv", BASE / "got_chunks.csv", BASE / "got_chunks.csv"]
VEC_PATHS = [BASE / "got_aug_vecs.npy", BASE / "got_vecs.npy", BASE / "got_chunks_vecs.npy"]
IDX_PATHS = [BASE / "got_aug_faiss.bin", BASE / "got_faiss.bin", BASE / "got_chunks_faiss.bin"]
TOK_PATHS = [BASE / "got_aug_corpus_tokens.pkl", BASE / "got_corpus_tokens.pkl", BASE / "got_corpus_tokens.pkl"]

# load dataframe (prefer augmented)
df_aug = None
for p in DF_AUG_PATHS + DF_CHUNK_PATHS:
    if p.exists():
        df_aug = pd.read_csv(p)
        print(f"Loaded dataframe from {p} ({len(df_aug)} rows)")
        break
if df_aug is None:
    raise FileNotFoundError("No chunk dataframe found. Expected one of: " + ", ".join(str(p) for p in DF_AUG_PATHS + DF_CHUNK_PATHS))

# load vectors & index
vec_path = next((p for p in VEC_PATHS if p.exists()), None)
idx_path = next((p for p in IDX_PATHS if p.exists()), None)

if vec_path is None and idx_path is None:
    raise FileNotFoundError("No precomputed vectors or FAISS index found in data/. Place got_aug_vecs.npy/got_aug_faiss.bin (or equivalents) in data/")

if vec_path is not None:
    vecs = np.load(vec_path)
    print(f"Loaded vectors from {vec_path} ({vecs.shape})")
else:
    vecs = None

if idx_path is not None:
    try:
        index = faiss.read_index(str(idx_path))
        print(f"Loaded FAISS index from {idx_path} (ntotal={index.ntotal})")
    except Exception as e:
        if vecs is None:
            raise RuntimeError(f"Failed to read FAISS index and no vectors to rebuild: {e}")
        faiss.normalize_L2(vecs)
        index = faiss.IndexFlatIP(vecs.shape[1])
        index.add(vecs)
        print("Rebuilt FAISS index from vectors")
else:
    # build index from vecs
    if vecs is None:
        raise FileNotFoundError("No FAISS index path and no vector file to build from.")
    faiss.normalize_L2(vecs)
    index = faiss.IndexFlatIP(vecs.shape[1])
    index.add(vecs)
    print("Built FAISS index from vectors (no index file present)")

# load BM25 token cache
tok_path = next((p for p in TOK_PATHS if p.exists()), None)
corpus_tokens = None
if tok_path:
    with open(tok_path, "rb") as f:
        corpus_tokens = pickle.load(f)
    print(f"Loaded BM25 token cache from {tok_path} ({len(corpus_tokens)} docs)")
else:
    raise FileNotFoundError("BM25 token cache not found (expected data/got_aug_corpus_tokens.pkl or similar). If you want to rebuild tokens run the tokenization cell once.")

# init BM25
bm25 = BM25Okapi(corpus_tokens)
print("BM25 index ready")

# set standard globals used by notebook cells
EMBED_MODEL = globals().get("EMBED_MODEL", "text-embedding-3-large")
_GOT_RETRIEVAL_SETUP_DONE = True
_bm25_row_count = len(df_aug)

# optional reranker placeholder (keeps behavior consistent)
RERANKER = None
try:
    from sentence_transformers import CrossEncoder
    RERANKER = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
    print("Loaded CrossEncoder reranker")
except Exception:
    print("CrossEncoder reranker not loaded (optional)")

print("\nSetup complete. Globals available: client, nlp, df_aug, index, bm25, EMBED_MODEL")
```

```
Loaded dataframe from data/got_augmented_lore.csv (17347 rows)
Loaded vectors from data/got_aug_vecs.npy ((17347, 3072))
Loaded FAISS index from data/got_aug_faiss.bin (ntotal=17347)
Loaded BM25 token cache from data/got_aug_corpus_tokens.pkl (17347 docs)
BM25 index ready
Loaded CrossEncoder reranker

Setup complete. Globals available: client, nlp, df_aug, index, bm25, EMBED_MODEL
```

Phase 3 — COSINE of Thrones Agentic RAG Scaffold

This cell defines a **custom, hand-built agentic RAG pipeline** that behaves like a mini-LangGraph workflow, but without using **LangChain**, **LangGraph**, or any external orchestration libraries. The entire graph is constructed manually using plain Python, dataclasses, and simple function composition.

What This Cell Implements

1. LangGraph-style State Object

A shared `RAGState` dataclass carries information through the pipeline:

- the user question
- detected question type

- named entities extracted from spaCy
- retrieved evidence chunks
- optional cross-encoder reranking
- formatted evidence text
- final grounded answer
- debug logs for each node

This mirrors LangGraph's idea of a mutable "state" flowing through nodes.

2. Four Agentic Nodes (Your Own Implementation)

The pipeline is built from four pure-Python nodes:

1. **Query Parser**
   Extracts named entities and question type (who/what/when/etc.).
   Adds structured metadata to the state for downstream routing.
2. **Retriever**
   Calls your **hybrid_search_aug** function which combines FAISS, BM25, and optional entity boosting over the augmented dataset (subtitles + lore).
   Stores the resulting dataframe into `state.retrieved`.
3. **Reranker (Optional)**
   If the cross-encoder model is available, it reranks chunks by semantic similarity, providing a higher-quality evidence list. Otherwise retrieval order is preserved.
4. **Synthesizer (LLM Agent)**
   Formats evidence and the question into a grounded, citation-first prompt.
   Calls OpenAI to produce a concise answer that only uses retrieved text.
   Writes the final answer into `state.answer`.

Each node is a plain Python function that takes and returns `RAGState`, exactly like a LangGraph node, but without relying on an external graph engine.

3. A Manual "Graph Runner"

The `run_graph(question)` function executes the pipeline in a fixed order:

```
# ==========================
# COSINE of Thrones — Phase 3 Agentic RAG Scaffold (LangGraph-style)
# ==========================
# Prereqs expected in memory:
# - nlp (spaCy English model)
# - client = OpenAI(...)
# - hybrid_search_aug(query, topk=10, alpha=0.6, cand_mult=20, use_ner_boost=True)
# - GEN_MODEL (e.g., "gpt-4o-mini")
# - ANSWER_PROMPT (or we define a concise one below)

from dataclasses import dataclass, field
from typing import List, Dict, Any, Optional
import pandas as pd
import numpy as np

GEN_MODEL = "gpt-4o-mini"

# ---- Optional cross-encoder reranker ----
RERANKER = None
try:
    from sentence_transformers import CrossEncoder
    RERANKER = CrossEncoder("cross-encoder/ms-marco-MiniLM-L-6-v2")
    print("Reranker loaded: cross-encoder/ms-marco-MiniLM-L-6-v2")
except Exception as _e:
    print("Reranker not available; continuing without cross-encoder.")

# ---- Minimal prompt (concise, citation-first) ----
ANSWER_PROMPT = """You are a Game of Thrones expert.
```

```
Answer in one sentence, using ONLY the evidence lines below. If evidence is insufficient, say:
"I cannot find this in the provided evidence."
Include season/episode if present.

Question: {question}

Evidence:
{evidence}
"""

# ---- Shared helpers ----
def format_evidence_rows(df: pd.DataFrame, k: int = 5) -> str:
    if df is None or len(df) == 0:
        return "(no evidence)"
    lines = []
    for _, r in df.head(k).iterrows():
        s, e = r.get("season"), r.get("episode")
        tag = f"S{int(s)}E{int(e)}" if pd.notna(s) and pd.notna(e) else "S?E?"
        txt = str(r.get("text","")).replace("\n"," ").strip()
        spk = r.get("speaker")
        prefix = f"[{tag}] {spk}: " if isinstance(spk, str) and spk.strip() else f"[{tag}] "
        lines.append(prefix + txt)
    return "\n".join(lines)

def guess_question_type(q: str) -> str:
    ql = q.lower()
    for t in ["who","what","when","where","why","how","which"]:
        if ql.startswith(t) or f" {t} " in ql:
            return t
    return "open"

def extract_entities(q: str) -> List[str]:
    doc = nlp(q)
    return [ent.text for ent in doc.ents]

# ---- Graph State ----
@dataclass
class RAGState:
    question: str
    question_type: str = "open"
    entities: List[str] = field(default_factory=list)
    retrieved: Optional[pd.DataFrame] = None
    reranked: Optional[pd.DataFrame] = None
    evidence_text: str = ""
    answer: str = ""
    logs: Dict[str, Any] = field(default_factory=dict)

# ---- Nodes ----
def node_query_parser(state: RAGState) -> RAGState:
    ents = extract_entities(state.question)
    qtype = guess_question_type(state.question)
    state.entities = ents
    state.question_type = qtype
    state.logs["parser"] = {"entities": ents, "question_type": qtype}
    return state

def node_retriever(state: RAGState, topk: int = 12, alpha: float = 0.6) -> RAGState:
    # Uses your augmented hybrid search
    hits = hybrid_search_aug(state.question, topk=topk, alpha=alpha)
    state.retrieved = hits
    state.logs["retriever"] = {"n_hits": 0 if hits is None else int(len(hits))}
    return state

def node_reranker(state: RAGState) -> RAGState:
    if state.retrieved is None or len(state.retrieved) == 0 or RERANKER is None:
        # fall back to retrieved as-is
        state.reranked = state.retrieved
        state.logs["reranker"] = {"used": False, "reason": "no-hits-or-no-model"}
        return state

    pairs = [[state.question, t] for t in state.retrieved["text"].tolist()]
    scores = RERANKER.predict(pairs)
    df = state.retrieved.copy().reset_index(drop=True)
    df["rerank_score"] = scores
    df = df.sort_values("rerank_score", ascending=False).reset_index(drop=True)
    state.reranked = df
    state.logs["reranker"] = {"used": True, "top_score": float(df.iloc[0]["rerank_score"])}
    return state

def node_synthesizer(state: RAGState, k_evidence: int = 5, show_prompt: bool = True) -> RAGState:
    """
    Generate grounded answer from retrieved evidence.
    Optionally prints the full prompt (question + evidence) before sending to OpenAI.
    """
    hits = state.reranked if state.reranked is not None else state.retrieved
    ev_text = format_evidence_rows(hits, k=k_evidence)
    state.evidence_text = ev_text

    # ---- Construct the full prompt ----
    prompt = ANSWER_PROMPT.format(question=state.question, evidence=ev_text)

    # ---- Print for debugging ----
    if show_prompt:
        print("\n=======================")
        print("🔍 PROMPT SENT TO OPENAI")
        print("=======================\n")
        print(prompt)
        print("\n=======================\n")

    # ---- Call the model ----
    out = client.chat.completions.create(
        model=GEN_MODEL,
        temperature=0.1,
        messages=[{"role": "user", "content": prompt}]
    )

    state.answer = out.choices[0].message.content.strip()
    state.logs["synthesizer"] = {
        "prompt_length_chars": len(prompt),
        "evidence_count": len(hits) if hits is not None else 0
    }
    return state

# ---- Graph runner ----
def run_graph(question: str) -> RAGState:
    state = RAGState(question=question)
    # Ordered pipeline
    state = node_query_parser(state)
    state = node_retriever(state, topk=12, alpha=0.6)
    state = node_reranker(state)
    state = node_synthesizer(state, k_evidence=5)
    return state

# ==========================
# Smoke test
# ==========================
sample_q = "Who is Jon Snow's mother in Game of Thrones?"
st = run_graph(sample_q)

print("Q:", st.question)
print("\n--- Evidence ---")
print(st.evidence_text)
print("\n--- Answer ---")
print(st.answer)
print("\n--- Logs ---")
for k,v in st.logs.items():
    print(k, "=>", v)
```

```
Reranker loaded: cross-encoder/ms-marco-MiniLM-L-6-v2

=======================
🔍 PROMPT SENT TO OPENAI
=======================
You are a Game of Thrones expert.
Answer in one sentence, using ONLY the evidence lines below. If evidence is insufficient, say:
"I cannot find this in the provided evidence."
Include season/episode if present.

Question: Who is Jon Snow's mother in Game of Thrones?

Evidence:
[S?E?] ===Series reprisals=== * Jon Snow (Kit Harington), a member of the Night's Watch and bastard son of Ned Stark. * Cersei Lannister (Lena Headey), the Queen Regent of the Seven Kingdoms serving
[S?E?] ===Jon Snow=== '''Jon Snow''' portrayed by Kit Harington. Kit Harington Jon Snow of House Stark and the Night's Watch is the secret son of Rhaegar Targaryen and Lyanna Stark, though raised as
[S7E3] Protector of the Seven Kingdoms, the Mother of Dragons, the Khaleesi of the Great Grass Sea, the Unburnt, the Breaker of Chains. This is Jon Snow.
[S?E?] === Jon Snow === Jon Snow was raised as Ned Stark's illegitimate son and serves as the point of view character in 42 chapters throughout ''A Game of Thrones'', ''A Clash of Kings'', ''A Storm
[S?E?] ===Parentage=== The identity of Jon's mother has created much speculation among readers of the series, and guessing her identity was the test Martin gave Benioff and Weiss when they approached


=======================

Q: Who is Jon Snow's mother in Game of Thrones?

--- Evidence ---
[S?E?] ===Series reprisals=== * Jon Snow (Kit Harington), a member of the Night's Watch and bastard son of Ned Stark. * Cersei Lannister (Lena Headey), the Queen Regent of the Seven Kingdoms serving
[S?E?] ===Jon Snow=== '''Jon Snow''' portrayed by Kit Harington. Kit Harington Jon Snow of House Stark and the Night's Watch is the secret son of Rhaegar Targaryen and Lyanna Stark, though raised as
[S7E3] Protector of the Seven Kingdoms, the Mother of Dragons, the Khaleesi of the Great Grass Sea, the Unburnt, the Breaker of Chains. This is Jon Snow.
[S?E?] === Jon Snow === Jon Snow was raised as Ned Stark's illegitimate son and serves as the point of view character in 42 chapters throughout ''A Game of Thrones'', ''A Clash of Kings'', ''A Storm
[S?E?] ===Parentage=== The identity of Jon's mother has created much speculation among readers of the series, and guessing her identity was the test Martin gave Benioff and Weiss when they approached

--- Answer ---
Jon Snow's mother is Lyanna Stark.

--- Logs ---
parser => {'entities': ["Jon Snow's", 'Thrones'], 'question_type': 'who'}
retriever => {'n_hits': 12}
reranker => {'used': True, 'top_score': 5.737795352935791}
synthesizer => {'prompt_length_chars': 6523, 'evidence_count': 12}
```